



AFRL-RI-RS-TR-2012-205

## **FOS: A FACTORED OPERATING SYSTEM FOR HIGH ASSURANCE AND SCALABILITY ON MULTICORES**

---

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

*AUGUST 2012*

FINAL TECHNICAL REPORT

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED*

STINFO COPY

**AIR FORCE RESEARCH LABORATORY  
INFORMATION DIRECTORATE**

## **NOTICE AND SIGNATURE PAGE**

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

AFRL-RI-RS-TR-2012-205 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

**/ S /**

CHRISTOPHER FLYNN  
Work Unit Manager

**/ S /**

PAUL ANTONIK  
Technical Advisor, Computing &  
Communications Division  
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

|   |                  |   |   |   |   |
|---|------------------|---|---|---|---|
| <b>REPORT DOCUMENTATION PAGE</b>  |                  |   |   | <b>Form Approved<br/>OMB No. 0704-0188</b>                                      |   |
| <small>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.</small>   |                  |   |   |   |   |
| <b>PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.</b>   |                  |   |   |   |   |
| <b>1. REPORT DATE (DD-MM-YYYY)</b><br>AUG 2012  |                  | <b>2. REPORT TYPE</b><br>FINAL TECHNICAL REPORT |   | <b>3. DATES COVERED (From - To)</b><br>MAR 2009 – MAR 2012                      |   |
| <b>4. TITLE AND SUBTITLE</b><br>FOS: A FACTORED OPERATING SYSTEM FOR HIGH ASSURANCE AND SCALABILITY ON MULTICORES   |                  |   |   | <b>5a. CONTRACT NUMBER</b><br>FA8750-09-1-0152                                  |   |
|   |                  |   |   | <b>5b. GRANT NUMBER</b><br>N/A  |   |
|   |                  |   |   | <b>5c. PROGRAM ELEMENT NUMBER</b><br>62788F                                     |   |
| <b>6. AUTHOR(S)</b><br>Anant Agarwal, Jason Miller, David Wentzlaff, Harshad Kasture, Nathan Beckmann, Charles Gruenwald III, and Christopher Johnson   |                  |   |   | <b>5d. PROJECT NUMBER</b><br>459T   |   |
|   |                  |   |   | <b>5e. TASK NUMBER</b><br>MI  |   |
|   |                  |   |   | <b>5f. WORK UNIT NUMBER</b><br>TF   |   |
| <b>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</b><br>Massachusetts Institute of Technology<br>77 Massachusetts Ave<br>Cambridge, MA 02139-4307  |                  |   |   | <b>8. PERFORMING ORGANIZATION REPORT NUMBER</b><br><br>N/A                      |   |
| <b>9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)</b><br>Air Force Research Laboratory/RITA<br>525 Brooks Road<br>Rome NY 13441-4505   |                  |   |   | <b>10. SPONSOR/MONITOR'S ACRONYM(S)</b><br>N/A                                  |   |
|   |                  |   |   | <b>11. SPONSORING/MONITORING AGENCY REPORT NUMBER</b><br>AFRL-RI-RS-TR-2012-205 |   |
| <b>12. DISTRIBUTION AVAILABILITY STATEMENT</b><br>Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.   |                  |   |   |   |   |
| <b>13. SUPPLEMENTARY NOTES</b>  |                  |   |   |   |   |
| <b>14. ABSTRACT</b><br>fos is a new operating system design for multicores and cloud computing. It builds on previous work in distributed and microkernel OSes by factoring services out of the kernel, and then further distributing each service into a parallel, distributed <i>fleet</i> of cooperating processes. This design naturally spans non-coherent shared memory architectures and clusters of machines. Additionally it provides increased isolation between and within services, giving opportunities for increased reliability. This report describes the general design principles of fos as well as the implementation of several specific services with evaluation of their scalability (e.g., naming, page allocation, and network stack). It also describes two distributed data structures (dPool and key-value store) that we implement to ease implementation of fos system services. |                  |   |   |   |   |
| <b>15. SUBJECT TERMS</b><br>fos, factored operating system, distributed operating system, high assurance, multicore   |                  |   |   |   |   |
| <b>16. SECURITY CLASSIFICATION OF:</b>  |                  |   | <b>17. LIMITATION OF ABSTRACT</b><br><br>UL | <b>18. NUMBER OF PAGES</b><br><br>27  | <b>19a. NAME OF RESPONSIBLE PERSON</b><br>CHRISTOPHER FLYNN |
| a. REPORT<br>U  | b. ABSTRACT<br>U | c. THIS PAGE<br>U                               |   |   | <b>19b. TELEPHONE NUMBER (Include area code)</b><br>N/A     |

## TABLE OF CONTENTS

|  |    |
|--|----|
| List of Figures .....  | ii |
| 1.0 Summary .....  | 1  |
| 2.0 Introduction.....  | 2  |
| 3.0 Methods, Assumptions, and Procedures (System Design) ..... | 4  |
| 3.1 Microkernel .....  | 4  |
| 3.2 Fleets .....   | 5  |
| 3.2.1 Scalability. ....  | 6  |
| 3.2.2 Self-awareness. ....                                     | 6  |
| 3.2.3 Elasticity. ....   | 7  |
| 3.2.4 Fault Tolerance. ....                                    | 7  |
| 4.0 Results and Discussion .....                               | 8  |
| 4.1 Summary of Accomplishments .....                           | 8  |
| 4.2 Messaging.....   | 9  |
| 4.3 Programming Model .....                                    | 10 |
| 4.4 Cloud Computing .....                                      | 11 |
| 4.5 Xen Paravirtualization.....                                | 11 |
| 4.6 Application Support .....                                  | 12 |
| 4.7 Services .....   | 13 |
| 4.7.1 Naming.....  | 13 |
| 4.7.2 Page Allocation.....                                     | 14 |
| 4.7.3 Network Stack.....                                       | 16 |
| 4.7.4 File System and Block Device.....                        | 17 |
| 4.7.5 Process Management. ....                                 | 17 |
| 4.7.6 Cloud Management.....                                    | 18 |
| 4.8 Distributed Data Structures .....                          | 18 |
| 4.8.1 dPool. ....  | 18 |
| 4.8.2 Key-Value Store.....                                     | 19 |
| 4.9 Tilera Port.....   | 19 |
| 5.0 Conclusions.....   | 19 |
| 6.0 References.....  | 21 |
| List of Symbols, Abbreviations and Acronyms .....              | 22 |

## LIST OF FIGURES

|   |    |
|---|----|
| Figure 1: Linux kernel memory allocation performance .....        | 3  |
| Figure 2: fos high-level architecture .....                       | 5  |
| Figure 3: Single-stream memcached latency .....                   | 12 |
| Figure 4: Scaling of the Name Service Fleet .....                 | 14 |
| Figure 5: Scalability of Physical Memory Allocation Service ..... | 15 |
| Figure 6: Comparison of Linux and fos page allocation.....        | 15 |
| Figure 7: fos Network Stack Design .....                          | 16 |
| Figure 8: Comparison of Linux and fos network scaling .....       | 17 |

## 1.0 SUMMARY

The next decade will bring single microprocessors containing 100's, 1000's, or even tens of 1000's of computing cores. While these processors will offer unprecedented quantities of computational resources, keeping all of those resources functioning properly will be a tremendous challenge. Besides the current problems of buggy software, future processors will experience increasing numbers of hard (permanent) and soft (transient) errors due to their smaller CMOS devices and increasing levels of integration. Contemporary operating systems have been designed to run on a small number of reliable cores and are not equipped to tolerate frequent errors. Managing 10,000 unreliable cores is so fundamentally different from managing two reliable cores that the fundamental design of operating systems and operating system data structures must be rethought.

Factored Operating System (fos) is a concept for a new operating system targeting 1000+ core multicore systems where space sharing replaces traditional time sharing to increase scalability and reliability. fos is built as a collection of Internet-inspired services. Each operating system service is factored into a set of communicating servers that, in aggregate, implement a system service. These servers, which are bound to dedicated cores, provide traditional kernel services and manage traditional kernel data structures in a factored, spatially-distributed manner. Running the servers on dedicated cores reduces the probability that they will be corrupted by buggy application code. Also, because they are spatially distributed, they provide a level of redundancy that allows the service to continue operating even if one or more server cores suffer errors. This is in stark contrast to current operating systems whose monolithic designs and shared central data structures create many opportunities for a single failure to cripple the entire system.

The fos project aims to build the prototypical open-source operating system for the 1000-core era. This includes a full suite of high-reliability system services that includes memory allocation, process management, protection, networking, and file-system services. Implementing a complete system for a simulated 1000-core microprocessor will allow us to verify the scalability of a factored design as well as experiment with different design choices and optimizations. When the implementation is sufficiently complete and stable, it will be released to the open source community to form the basis for future OS research and development.

This report contains our efforts to construct the system, both the general design principles of fos's scalable service model and implementations of specific services. We describe the design of naming, page allocation, network stack, file system, and process management services including experimental evaluation of the naming, page allocation, and network stack. These results indicate that fos compares well against Linux, even when accounting for cores devoted to the OS. We discuss how fos's fleet design gives opportunities for improved reliability in several services and spanning the full system stack. We also describe two generalized distributed data structures (dPool and key-value store) that we have implemented to ease the implementation fos services.

## 2.0 INTRODUCTION

The number of processing cores on single-chip microprocessors is increasing rapidly. The recent shift from single-stream to multicore processor designs is motivated by an inability to maintain exponential performance improvement in single-stream designs. Because this shift is out of necessity rather than choice, it is likely to continue for the foreseeable future [1]. Extrapolating current growth rates, a single microprocessor will contain between 1,000 and 10,000 cores within the next 10 years [2].

Unfortunately, current operating systems are incapable of dealing with the realities of future multicore systems. They were designed for single-processor computers and adapted to handle systems with small numbers of cores. In contrast to previous hardware generations, where additional resources were hidden behind abstraction layers and ISAs, multicore processors expose new resources in the form of additional cores and require the software to decide how to manage them. The task of managing 10,000 cores is so fundamentally different from the task of managing two cores that the entire design of operating systems must be rethought.

One of the key problems in managing large-scale multicores is reliability. As CMOS technology advances and devices become smaller they are more susceptible to manufacturing defects and transient external interferences (*e.g.*, cosmic rays) [3, 4]. This increases the probability that a particular device will experience a failure. At the same time, the number of these devices on each chip is increasing exponentially. Therefore, future chips will experience many more permanent and transient errors than current processors. The good news is that multicore designs naturally segment a chip's resources so that a single error will probably only affect one core. The bad news is that current operating systems rely on many centralized global structures such that an error in even a single core can corrupt the entire system.

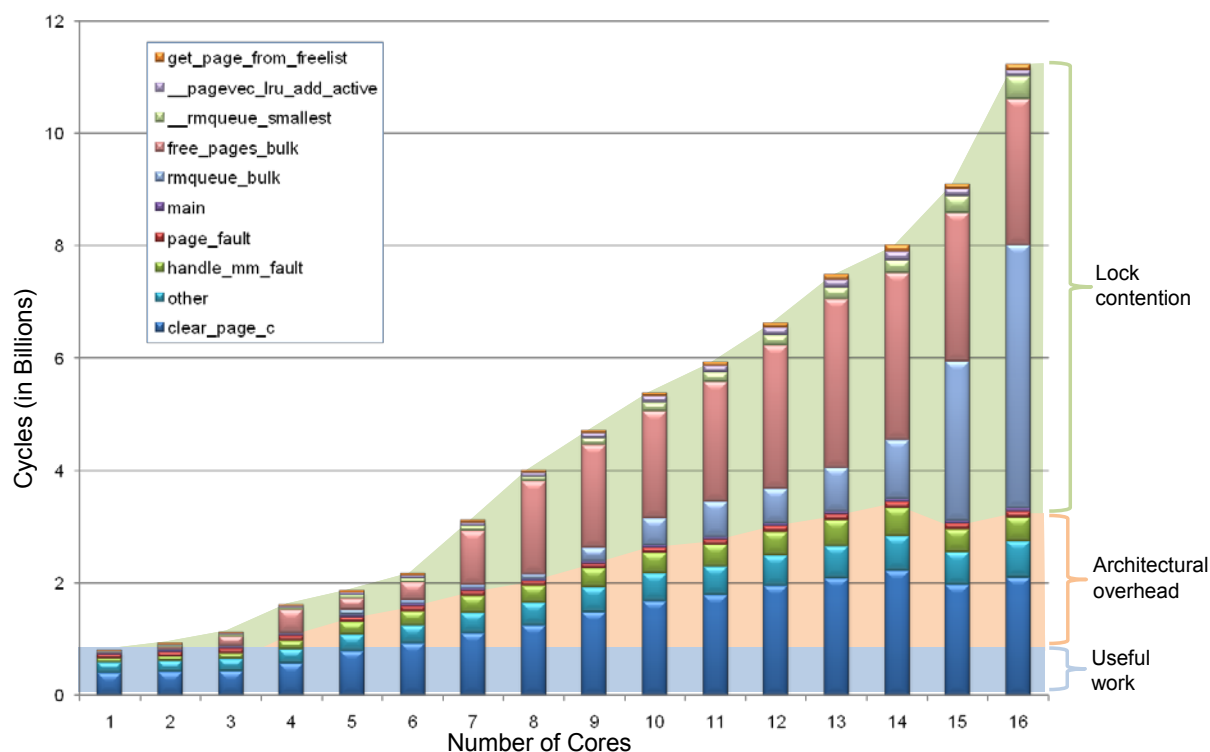
The problem with global data structures in multicore chips is that they do not scale well, thereby creating single points of failure and performance bottlenecks [2]. To make effective use of thousands of cores, future operating systems will need to address the issue of scalability. Current symmetric multiprocessor (SMP) operating systems have been designed to manage a small number of cores. With multicore chips, the number of cores will be increasing at an exponential rate. Any OS designed to run on them will need to embrace scalability and make it a first-order design constraint for reasons of both reliability and performance.

Contemporary operating systems for multiprocessor computers have evolved from uniprocessor operating systems. As a result, they have several characteristics that prevent them from scaling to 1000-core systems. Two of the biggest problems are centralized data structures protected by locks and reliance on efficient hardware shared memory.

The initial approach to adapting uniprocessor operating systems to parallel machines was to add a single large lock protecting the entire kernel. This prevents multiple threads from simultaneously entering the kernel and therefore preserves the invariant that all kernel data structures are accessed by one thread at a time. Unfortunately, a single kernel lock, by definition, limits the concurrency achievable within an OS kernel and hence the scalability. The traditional method of improving scalability has been to successively create finer-grained locks thus reducing the probability that more than one thread is attempting to concurrently access locked data. However, this approach suffers from two problems. First, adding locks into an operating system is a very time consuming and error prone endeavor. These errors are frequently subtle and go unnoticed during normal testing; only exposing themselves in unusual circumstances. Second, each lock is manipulated by all the cores and is ultimately stored in a

single location. A failure in either a core using the lock or the storage location can result in either unsafe parallel execution or complete deadlock of the entire system.

Figure 1 demonstrates the reliance of current operating systems on centralized structures. It shows the performance impact of centralized locks in the memory allocation routines of the Linux kernel. This data was collected from the Linux 2.6.24.7 kernel, running on a 16-core Intel machine, using a synthetic app designed to stress the memory allocation system. As the number of cores attempting to allocate memory increases, the amount of time wasted on lock contention dominates all other factors. It is clear that the existing kernel does not scale well beyond eight cores, despite the fact that these routines have already been extensively optimized using fine-grained locks. It is also clear that locks are heavily used, thereby creating many opportunities for a lock-related failure to bring down the system.



**Figure 1: Linux kernel memory allocation performance**

Aside from the difficulties with locks, contemporary operating systems are hampered by their reliance on shared memory for communication between cores. This is largely due to the fact that shared memory is the only communication mechanism provided by current machines. However, it is doubtful that future large-scale multicores will be able to provide efficient full-machine cache coherence as the abstraction of a globally shared memory space is inherently a shared global structure. Even if they could, a single failure in the “home” node for a memory location could corrupt or disable communication between many other cores.

### **3.0 METHODS, ASSUMPTIONS, AND PROCEDURES (SYSTEM DESIGN)**

Current OSes were designed in an era when computation was a limited resource. With the expected exponential increase in number of cores, the landscape has fundamentally changed. The question is no longer how to cope with limited resources, but rather how to make the most of the abundant computation available. fos is designed with this in mind, and takes scalability and adaptability as the first-order design constraints. The goal of fos is to design system services that scale from a few to thousands of cores.

fos does this by factoring OS services into userspace processes, running on separate cores from the application. Traditional monolithic OSs time multiplex the OS and application, whereas fos spatially multiplexes OS services (running as user processes) and application processes. In a regime of one to a few cores, time multiplexing is an obvious win because processor time is precious and communication costs are low. With large multicores and the cloud, however, processors are relatively abundant and communication costs begin to dominate. Running the OS on every core introduces unnecessary sharing of OS data and associated communication overheads; consolidating the OS to a few cores eliminates this. For applications that do not scale well to all available cores, factoring the OS is advantageous in order to accelerate the application. In this scenario, spatial scheduling (layout) becomes more important than time multiplexing within a single core.

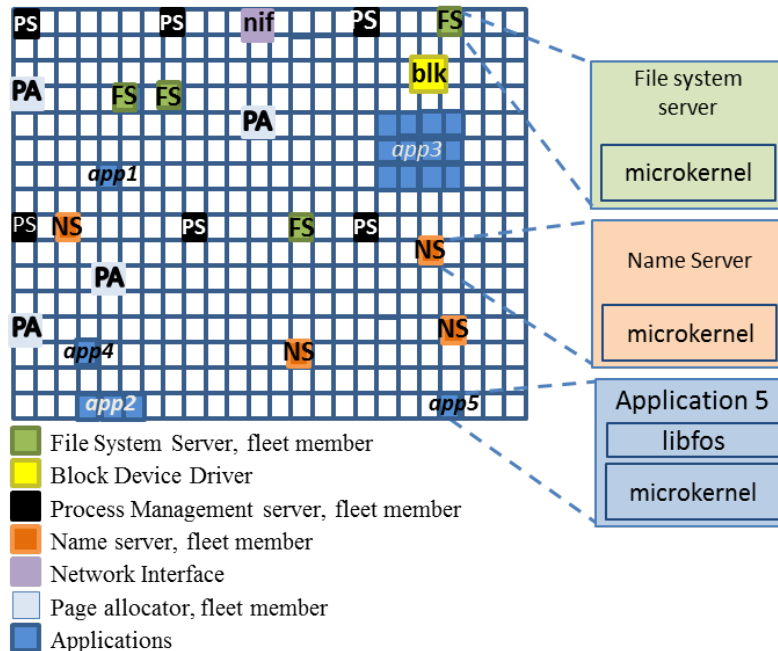
However, even when the application could consume all cores to good purpose, running the OS on separate cores from the application provides a number of advantages. Cache pollution from the OS is reduced, and OS data is kept hot in the cache of those cores running the service. The OS and the application can run in parallel, pipelining OS and application processing, and often eliminating expensive context switches. Running services as independent threads of execution also enables extensive background optimizations and re-balancing. Although background operations exist in monolithic OSes, fos facilitates such behavior since each service has its own thread of control.

In order to meet demand in a large multicore or cloud environment, reduce access latency to OS services and increase throughput, it is necessary to further parallelize each service into a set of distributed, cooperating servers. We term such a service a fleet.

Figure 2 shows the high-level architecture of fos. A small microkernel runs on every core. Operating system services and applications run on distinct cores. Applications can use shared memory, but OS services communicate only via message passing. A library layer (libfos) translates traditional syscalls into messages to fos services. A naming service is used to find a message's destination server. The naming service is maintained by a fleet of naming servers. Finally, fos can run on top of a hypervisor and seamlessly span multiple machines, thereby providing a single system image across a cloud computer.

#### **3.1 Microkernel**

In order to factor OS services into fleets, fos uses a minimal microkernel design. The microkernel provides only: (i) a protected messaging layer, (ii) a name cache to accelerate message delivery, (iii) rudimentary time multiplexing of cores, and (iv) an application programming interface (API) to allow the modification of address spaces and thread creation. All other OS functionality and applications execute in user space. However, many OS system services possess special capabilities that grant them privileges beyond those of regular applications.



**Figure 2: fos high-level architecture**

Capabilities are extensively used to restrict access into the protected microkernel. For instance, the memory modification API allows a process on one core to modify the memory and address space on another core if appropriate capabilities are held. This approach allows fos to move significant memory management and scheduling logic into userland processes. Capabilities are also used in the messaging system to determine who is allowed to send messages to whom.

### 3.2 Fleets

This section discusses how fos supports building fleets, and the principles used in building them. The programming model used to construct fleets is also discussed, highlighting the tools and libraries provided by fos to ease their construction.

Services in fos are implemented by cooperating, spatially-distributed sets of processes. This idea is the cornerstone of fos. Whereas prior projects have demonstrated the viability of microkernels, fos aims to implement a complete distributed, parallel OS by implementing service fleets. The core design principles of fleets are:

- **Scalability.** Fleets are designed with scalability as the primary design constraint. Fleets employ best practices for scalability such as lockless design and data partitioning, as well as the best available data structures and algorithms.
- **Self-awareness.** Fleets monitor and adapt their behavior to the executing environment. Load between members is rebalanced, and members are migrated to improve communication latency.
- **Elasticity.** Fleets are elastic, and can expand and shrink to match changing demand. Performance is monitored such that the optimal number of servers is used to implement each OS service.

- **Fault Tolerance.** Fleets are naturally tolerant to faults, as they do not share memory and therefore have a much higher degree of isolation than conventional OSes.

Each system service is implemented by a single fleet of servers. Within a single system, there will be a file system fleet, a page allocator fleet, a naming fleet, a process management fleet, etc. Additionally, the fleet may span multiple machines where advantageous. For example, in order to provide local caching for fast access, it is good practice to have a member of the file system fleet on every machine. The same general principle applies to many OS services, and for some critical services (e.g. naming) it is required to have an instance on each machine.

Fleets must support a variety of management tasks. Fleets can grow and shrink to meet demand, and must support rebalancing when a new member joins or leaves the fleet. Currently many services designate a single member, termed the coordinator, to perform many of these tasks.

**3.2.1 Scalability.** Fleets are designed to scale from a few to very many servers. They are not tuned to a particular size, but designed using best practices and algorithms to scale over a large range of sizes. This is important for multicore and cloud computing, as current trends in increasing core counts are likely to continue for the foreseeable future. Furthermore, different processors, even within a single processor family, will have variety of core counts. Therefore, fleets are designed to scale to different number of cores to address these needs.

In order to facilitate the scalability of fos fleets, fleets are designed in a message-passing-only manner such that layout of the data is explicit and shared memory and lock contention do not become bottlenecks. Our results show that lock contention in Linux has major scalability impact on the page allocation service, whereas fos is able to achieve excellent scalability through lockless design.

**3.2.2 Self-awareness.** A natural advantage of separating OS services from applications is the ease of performing background optimizations and re-balancing of the service. Although such optimizations are possible in monolithic designs, giving each service its own thread provides a natural framework in which to perform such tasks. Interference with application performance can be minimized by performing tasks only when necessary or when the service is idle. Fleets monitor their environment and adapt their behavior to improve performance. For example, fleet members can migrate to minimize communication costs with cores they are serving. Similarly, when a new transaction begins, it is assigned to the closest available fleet member. Active transactions can be migrated to other members if a server becomes overloaded, and these performance statistics also motivate growing or shrinking the fleet.

Fleets often must route requests themselves, independent of the name service. One important reason is resource affinity – if a request uses a resource under management of a particular fleet member, then the request should be forwarded to that member. A simple example of this is local state kept by each fleet member for a transaction, for example a Transmission control Protocol/Internet Protocol (TCP/IP) connection. In this case, routing through the name service is insufficient because state has already been created during connection establishment, and the connection is associated with a particular fleet member when the first message on that connection arrives (see Section 4.1). Another example is if a request uses a hardware resource on a different machine. In this case, the request must be forwarded to the fleet member on the machine that has access to the hardware.

**3.2.3 Elasticity.** In addition to unprecedented amounts of resources, clouds and multicores introduce unprecedented variability in demand for these resources. Dynamic load balancing and migration of processes go a long way towards solving this problem, but still require over-provisioning of resources to meet demand. This would quickly become infeasible, as every service in the system claims the maximum amount of resources it will ever need. Instead, fleets are elastic, meaning they can grow to meet increases in demand, and then shrink to free resources back to the OS.

Monolithic OSes achieve elasticity by “accident”, as OS code runs on the same core as the application code. This design has obvious advantages, since the computational resources devoted to the service scale proportionally with demand. There are disadvantages, however: monolithic designs relinquish control of how many cores to provision the service. This can lead to performance degradation if too many threads are accessing a shared resource simultaneously. fos can avoid this by fixing the size of a fleet at the point that achieves maximal performance. One example of “elasticity by accident” running awry occurs when a single lock is highly contended. In this case, when more cores contend for a lock, the performance of all cores degrades. Limiting the numbers of cores performing OS functions (contending for the resource) can actually improve performance in such cases. Our results show examples of this phenomenon where by limiting the number of cores dedicated to a fleet, fos can achieve higher performance with fewer resources than Linux simply because Linux has no means to limit the number of cores running the OS services.

Additionally, for applications that rely heavily on the OS it may be best to provision more cores to the OS service than the application. The servers can then collaborate to provide the service more efficiently. These design points are not provided in monolithic operating systems.

A fleet is grown by starting a new server instance on a new core. This instance joins the fleet by contacting other members (either the coordinator or individual members via a distributed discovery protocol) and synchronizing its state. Some of the distributed, shared state is migrated to the new member, along with the associated transactions. Transactions are migrated in any number of ways, for example by sending a redirect message to the client from the “old” server. Shrinking the fleet can be accomplished in a similar manner.

**3.2.4 Fault Tolerance.** The fleet design has natural advantages for fault tolerance. Because shared state is managed through a library of distributed data structures, there are natural opportunities for replication of critical data. Memory is not shared, so a faulty core cannot corrupt the memory of an entire service or, worse, the full system. Because each server keeps its own local state, it is not affected by misbehavior of other fleet members for many operations.

For example, the name service fully replicates the name table on all members, and there is no central coordinator or point of failure. This design is naturally tolerant of faults in any member. Applications communicating to a faulty name service may get incorrect results, but the namespace remains operational and applications can detect failure upon use of invalid names and switch to a non-faulty name service member.

## **4.0 RESULTS AND DISCUSSION**

### **4.1 Summary of Accomplishments**

This subsection summarizes the accomplishments achieved by this project. They are discussed in further detail later in this section.

The fos microkernel was developed with support for large x86 systems. fos is implemented as a paravirtualized OS on Xen to support cloud systems. The OS has support for large multicore systems (above 32 cores), and it has driver support for Ethernet and block devices under Xen.

fos's messaging system has gone through several iterations to support performance and transparency across different mechanisms. This involved implementing multiple messaging transports, a proxy service for inter-machine communications, and a naming service to support discovery of other services. The fast-path channel messaging implementation also went through several iterations to achieve better performance.

We designed and implemented a service programming model for fos. This includes a lightweight cooperative threading library, a dispatch mechanism, and an RPC stub generation tool. This model is used by all fleets.

We designed and implemented distributed data structures to be used by system services. A distributed pool data structure is used in the page allocator and process management service to allocate from a pool of homogeneous object (memory pages or PIDs). The service performs background rebalancing for performance. A distributed key-value store is used by the name service to store the name space. This data structure is completely distributed without a central coordinator or point of failure. It is also completely replicated for fault tolerance and read latency.

We implemented several key fleets, including the page allocator, name service, network stack, file system, process manager, and cloud manager (via Eucalyptus).

Several real-world benchmarks and workloads are supported by fos, including lighttpd [5], memcached [6], SQLite [7], SPLASH [8], and PARSEC [9].

fos services have been extensively evaluated in terms of raw performance as well as scalability. This includes baseline measurements of messaging, "null system call," and single-core benchmarking of each service, as well as scaling studies of services as cores are added to the system.

A port of fos to the Tilera multicore system was in progress at the time this report was being written.

## 4.2 Messaging

fos provides interprocess communication through a mailbox-based message-passing abstraction. The Application Programming Interface (API) allows processes to create mailboxes to receive messages, and associate the mailbox with a name and capability. This design provides several advantages for a scalable OS on multicores and in the cloud. Messaging can be implemented via a variety of underlying mechanisms: shared memory, hardware message passing, TCP/IP, etc. This allows fos to run on a variety of architectures and environments.

The traditional alternative to message-passing is shared memory. However, in many cases shared memory may be unavailable or inefficient: fos is architected to support unconventional architectures where shared memory support is either absent or inefficient, as well as supporting future multicores with thousands of cores where global shared memory may prove unscalable. Relying on messaging is even more important in the cloud, where machines can potentially reside in different datacenters and intermachine shared memory is unavailable.

A more subtle advantage of message passing is the programming model. Although perhaps less familiar to the programmer, a message-passing programming model makes data sharing more explicit. This allows the programmer to consider carefully the data sharing patterns and find performance bottlenecks early on. This leads to more efficient and scalable designs. Through message passing, we achieve better encapsulation as well as scalability. It bears noting that fos supports conventional multithreaded applications with shared memory, where hardware supports it. This is in order to support legacy code as well as a variety of programming models. However, OS services are implemented strictly using messages.

Having the OS provide a single message-passing abstraction allows transparent scale-out of the system, since the system can decide where best to place processes without concern for straddling shared memory domains as occurs in cloud systems. Also, the flat communication medium allows the OS to perform targeted optimizations across all active processes, such as placing heavily communicating processes near each other.

fos currently provides three different mechanisms for message delivery: kernelspace, userspace, and intermachine. These mechanisms are transparently multiplexed in the libfos library layer, based on the locations of the processes and communication patterns:

- **Kernelspace:** The fos microkernel provides a simple implementation of the mailbox API over shared memory. This is the default mechanism for delivering messages within a single machine. Mailboxes are created within the address space of the creating process. Messages are sent by trapping into the microkernel, which checks the capability and delivers the message to the mailbox by copying the message data across address spaces into the receiving process. Messages are received without trapping into the microkernel by polling the mailbox's memory. The receiver is not required to copy the message a second time because the microkernel is trusted to not modify a message once it is delivered.
- **Userspace:** For processes that communicate often, fos also provides shared memory channel-based messaging inspired by URPC [10] and Barrelfish [11]. The primary advantage of this mechanism is that it avoids system call overhead by running entirely in user space. Channels are created and destroyed dynamically, allowing compatibility with fos's mailbox messaging model. Outgoing channels are bound to names and stored in a user-level name cache. When a channel is established, the microkernel maps a shared page between the sender and receiver. This page is treated

as a circular queue of messages. Data must be copied twice, once by the sender when the message is enqueued in the buffer and once by the receiver when the message is dequeued from the buffer. The second copy is needed for security and to ensure the queue slot is available for future messages as soon as possible. This mechanism achieves much better per-message latency, at the cost of an initial overhead to establish a connection.

- Intermachine: Messages sent between machines go through a proxy server. This server is responsible for routing the message to the correct machine within the fos system, encapsulating messages in TCP/IP, as well as maintaining the appropriate connections and state with other proxy servers in the system.

### 4.3 Programming Model

fos provides libraries and tools to ease the construction of fleets and parallel applications. These are designed to mitigate the complexity and unfamiliarity of the message-passing programming paradigm, thus allowing efficient servers to be written with simple, straight-line code. These tools are (i) a cooperative threading model integrated with fos's messaging system, (ii) a remote procedure call (RPC) code generation tool, and (iii) a library of distributed objects to manage shared state.

The cooperative threading model and RPC generation tool are similar to tools commonly found in other OSes. The cooperative threading model lets several active contexts multiplex within a single process. The most significant feature of the threading model is how it is integrated with fos's messaging system. The threading model provides a dispatcher, which implements a callback mechanism based on message types. When a message of a particular type arrives, a new thread is spawned to handle that message. Threads can send messages via the dispatcher, which sleeps the thread until a response arrives. The use of a cooperative threading model allows the fleet server writer to write straight-line code for a single transaction and not have to worry about preemptive modification of data structures thereby reducing the need for locks. The RPC code generator provides the illusion of local function calls for services implemented in other processes. It parses regular C header files and generates server and client-side libraries that marshal parameters between servers. The tool parses standard C, with some custom annotations via gccxml indicating the semantics of each parameter. Additionally, custom serialization and deserialization routines can be supplied to handle arbitrary data structures. The RPC tool is designed on top of the dispatcher, so that servers implicitly sleep on a RPC call to another process until it returns.

The libraries generated by the RPC tool provide more general support for constructing fleets. For example, they can be used to pack and unpack messages without RPC (send, sleep, return) semantics. This is useful in order to pipeline requests with additional processing, broadcast a message to fleet members, and support unusual communication patterns that arise in constructing fundamental OS services.

One challenge to implementing OS services as fleets is the management of shared state. This is the major issue that breaks the illusion of straight-line code from the RPC tool. fos addresses this by providing a library of distributed data structures that provide the illusion of local data access for distributed, shared state. The goal of this library is to provide an easy way to distribute state while maintaining performance and consistency guarantees. Data structures are provided matching common usage patterns seen in the implementation of fos services. The name service provides a distributed key-value store, implemented via a two-phase commit protocol

with full replication. The library provides another key-value store implementation that distributes the state among participants. These implementations have different cost models, and usage dictates when each is appropriate. Similarly, the page allocator uses a distributed buddy allocator; this data structure could be leveraged to provide process IDs, file pointers, etc. These data structures are kept consistent using background updates. This is achieved using the cooperative dispatcher discussed above. The distributed data structure registers a new mailbox with the dispatcher and its own callbacks and message types.

Future research directions will explore the set of data structures that should be included in this library, and common paradigms that should be captured to enable custom data structures.

## **4.4 Cloud Computing**

We realized that many of the features of fos that were designed for large-scale multicore systems were also useful in a cloud environment. Because of fos' distributed nature, scalability, and fault tolerance it can easily be extended to work across multiple machines in addition to multiple cores. Therefore, we have expanded our strategic vision of fos from an operating system for multicore chips to a unified operating system for large numbers of cores either within a single machine or spread across many machines. We do not expect this to significantly change the design we have already created; however, future research will include developing additional features for cloud support.

fos has been expanded to support dynamic joining of new instances to a running instance. Fos initially required all machines to be pre-allocated and initialized simultaneously. We expanded the multi-machine support to allow dynamic joining of new instances. This allows new machines to be added for increased capacity/performance or to take over from failed machines. This new functionality requires merging state of services on the new instances so they have a consistent view of the instance. For example, when a new machine joins the instance, the name service performs a distributed lock that blocks further modification to the name space. Then the name table is duplicated on the new instance, the lock is released, and normal operation resumes with the new machine fully joined in the namespace.

## **4.5 Xen Paravirtualization**

We have decided to implement our initial prototype version of fos as a paravirtualized OS for Xen instead of a bare-metal OS that runs directly on hardware. Essentially, this means that we will be using an off-the-shelf hypervisor layer (Xen) rather than creating our own. All of the fos-specific low-level functionality will be moved to the microkernel layer than sits on top of the hypervisor.

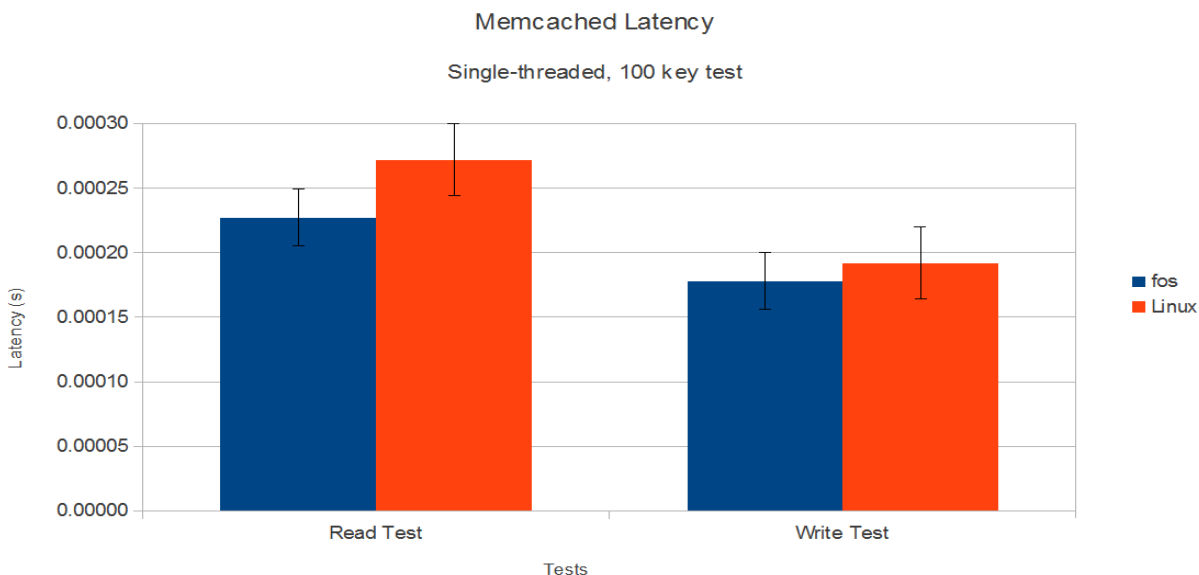
This approach has several benefits. First, it allows us to get a working system up and running more quickly because we have less to implement ourselves. Second, it allows us to leverage existing Linux device drivers to quickly run on a broader range of hardware. Xen provides several types of virtual devices with simplified interfaces and uses Linux devices drivers running in a separate virtual machine to bridge between the virtual devices and real hardware. Therefore, we only need to write a single device driver for each type of device (*e.g.*, network interface, video, disk controller, etc.) and we will be able to run on any of those devices that Linux supports. Third, it allows us to run experiments on commercially-available cloud infrastructures. In particular, Amazon's EC2 uses Xen for virtualization and requires that any custom virtual machines use paravirtualization.

It is important to note that the decision to use Xen paravirtualization does not limit the long-term portability of fos. To implement a bare-metal version or port to another architecture, we will simply need to implement our own hypervisor layer and device drivers to replace Xen. This is work we would have needed to do anyway, we have simply delayed the point at which it is required and allowed ourselves to get to the more interesting aspects of fos earlier.

## 4.6 Application Support

fos supports a number of important cloud and multicore workloads. It implements the commonly-used APIs in POSIX threads, POSIX sockets, libc, etc. through a compatibility layer above libfos. The multi-threading work involved evaluating our pthreads library along with its kernel extensions and porting several pthreads applications to fos, such as parallel pthread versions of Jacobi relaxation, matrix-matrix multiply, 2D and 3D molecular dynamics codes as well as several mutex stress codes and threading bombs. We have done several scalability evaluations for our threaded library implementation using these multithreaded codes, which has guided our optimization efforts for our library implementation. The results of our optimization efforts for fos pthread library has achieved similar scalability and performance numbers for most pthread operations (except for thread creation and joining) as the Linux 2.6 pthreads library implementation.

Figure 3 shows that fos achieves competitive performance on memcached for small workloads. This benchmark measures the latency of memcached requests through fos and Linux over 100 requests. It exercises the full network stack and POSIX compatibility layer. It is ongoing work to scale our network stack and application support to larger memcached workloads.



**Figure 3: Single-stream memcached latency**

## 4.7 Services

**4.7.1 Naming.** Closely coupled with messaging, fos provides a name service to lookup mailboxes throughout the system. Each name is a hierarchical URI much like a web address or filename. The namespace is populated by processes registering their mailboxes with the name service. The key advantage of the name service is the level of indirection between the symbolic identifier of a mailbox and its so-called “address” or actual location (machine, memory address, etc.). By dealing with names instead of addresses, the system can dynamically load balance as well as re-route messages to facilitate and processes migration.

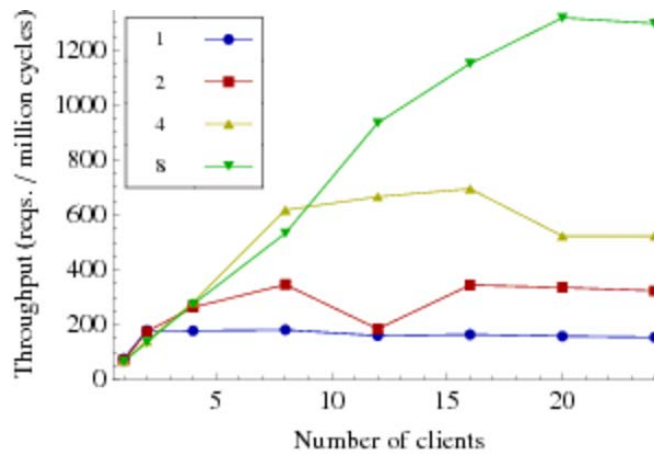
The need for dynamic load balancing and process migration is a direct consequence of the massive scale of current cloud systems and future multicores. In addition to a greater amount of resources under management, there is also greater variability of demand. Static scheduling is inadequate, as even if demand is known it is rarely constant. It is, therefore, necessary to adapt the layout of processes in the system to respond to where the service is currently needed.

The advantage of naming is closely tied to fleets. Fleet members will each have an in-bound mailbox upon which they receive requests, and these mailboxes will all be registered under a single name. It is the responsibility of the name service to resolve a request to one member of a fleet. Load balancing can be quite complicated and highly customized to a specific service. Each service can dynamically update the name system to control the load balancing policy for their fleet. The name service does not determine load balancing policy, but merely provides mechanisms to implement a policy. To support stateful operations, applications or libfos can cache the name lookups so that all messages for a transaction go the same fleet member.

Alternatively, the fleet can manage shared state so that all members can handle any request. In fos, another design point is to explicitly load balance within the fleet. This approach may be suitable when the routing decision is based on state information not available to the name service. In either approach it is important to realize that by decoupling the lookup of mailboxes using a symbolic name, the OS has the freedom to implement a given service through a dynamic number of servers. For example, the name lookup of /foo/bar results in the symbolic name /foo/bar/3, which is the third member of the fleet. This is the name that is cached, and subsequent requests forward to this name, wherever it should be.

The name service also enables migration of processes and their mailboxes. This is desirable for a number of reasons, chiefly to improve performance by moving communicating servers closer to each other. Migration is also useful to rebalance load as resources are freed. The name service provides the essential level of indirection that lets mailboxes move freely without interrupting communication.

Figure 4 shows the scaling of the name service fleet with a read-dominated workload. Name service fleets of appropriate size scale ideally, with minimal performance degradation under high load. This workload exercises the critical operation of the name service – lookups to other OS services, where the name service is on the critical path.



**Figure 4: Scaling of the Name Service Fleet**

**4.7.2 Page Allocation.** We have implemented a new parallel version of the Physical Memory Allocation (PMA) service using the dPool data structure and used it to evaluate the performance of the different dPool implementations. The PMA maintains a pool of available memory pages and responds to requests for more memory from running processes. Of course, it is also possible for processes to release pages back to the PMA when they no longer need them. The dPool data structure is used to maintain the list of available pages. Processes can request new pages from any of the servers within the PMA fleet but will typically use the server closest to themselves (based on communication latency). This will minimize communication time and spread the total system load across the different servers. Although we have not yet implemented this, it should also be possible to redirect processes from one server to another if the load is highly imbalanced and a particular server becomes a bottleneck.

Figure 5 shows performance results for the PMA when using the Background Push with Estimation version of the dPool. This was the best performing dPool implementation for this service. The different lines show the achieved response rate of the PMA service for different fleet sizes (similar data from a stock Linux installation on the same hardware is also included for reference). In this experiment, the fleet size did not change dynamically but was fixed while the number of requesting clients was varied. The total memory request rate increases in proportion to the number of clients. In all cases, the PMA fleet's response rate scales nicely as the load increases, up to some saturation point. This shows the maximum load that a particular fleet size can support. This maximum also scales linearly as the number of servers in the fleet increases. From this we conclude that the dPool data structure is able to manage the shared state for the PMA service without introducing any scaling bottlenecks.

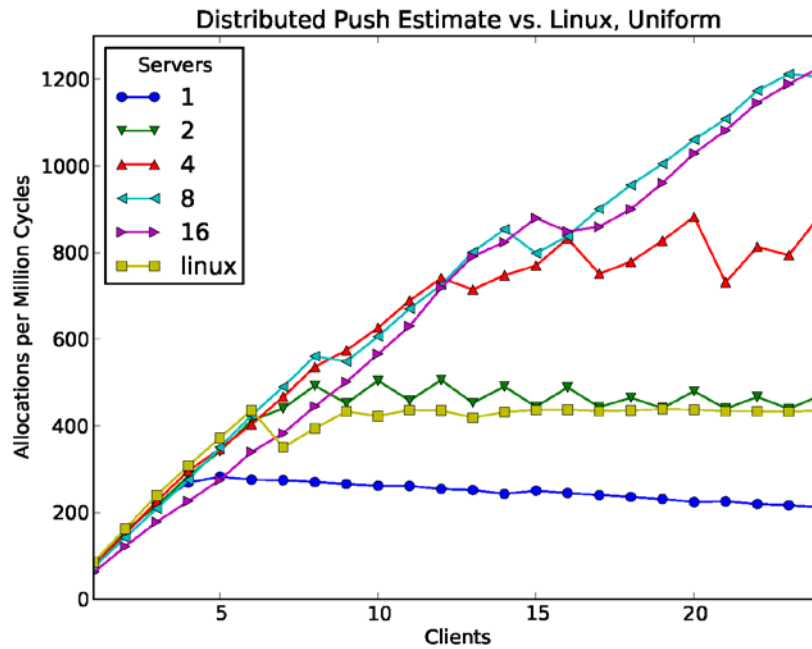


Figure 5: Scalability of Physical Memory Allocation Service

Figure 6 shows a direct comparison of fos' page allocation service to Linux's kernel allocator versus the number of cores in the system. This accurate represents the overhead of fleets, and shows that despite dedicating cores exclusively to the OS, fos still significantly outperforms Linux's page allocator.

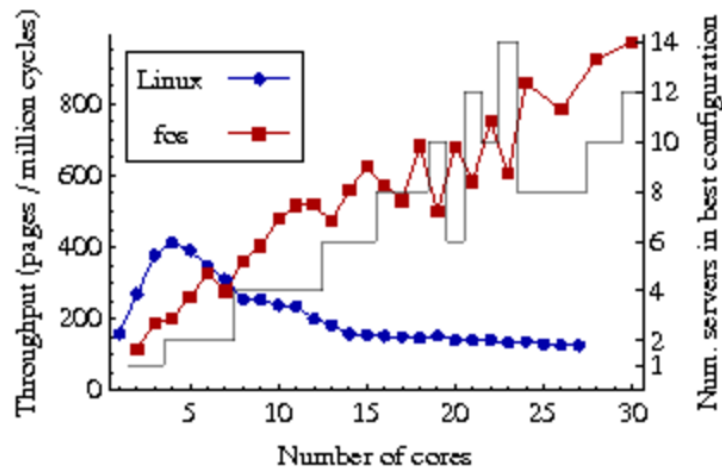
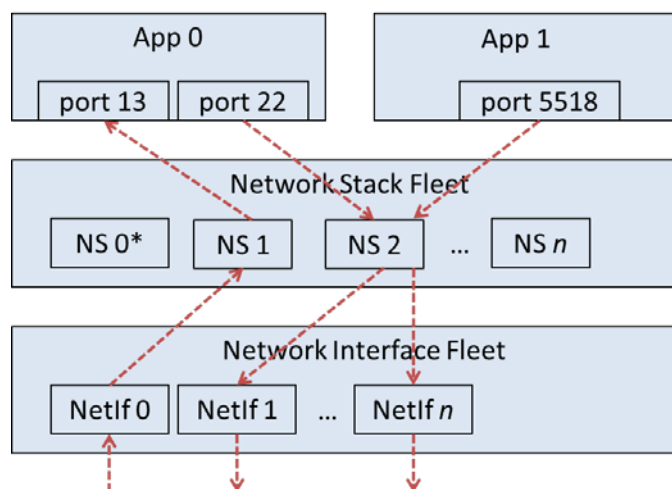


Figure 6: Comparison of Linux and fos page allocation

**4.7.3 Network Stack.** fos has a fully-featured networking service responsible for packing and unpacking data for the various layers of the network stack as well as updating state information and tables associated with the various protocols (e.g., Dynamic Host Configuration Protocol (DHCP), Address Resolution Protocol (ARP), and Domain Name System (DNS)). The stack was implemented by extending lwIP with fos primitives for parallelization to create the network stack fleet. The logical view of this service is depicted in Figure 7. In this diagram the dashed lines represent the paths that a given TCP/IP flow may take while traversing the network stack. In this diagram we can see that the flows are multiplexed between the different network stack fleet members. The distribution of these flows amongst the fleet members is managed by the fleet coordinator.



\* Netstack 0 acts as the coordinator.

**Figure 7: fos Network Stack Design**

The design employs a fleet of network stack servers with a single member designated as the coordinator. The fleet coordinator is responsible for several management tasks as well as handling several of the protocols.

When the kernel receives data from the network interface it delivers it to the network interface server. The network interface server then peeks into the packet and delivers it to one of the fleet members depending on the protocol the packet corresponds to. The handling of many stateless protocols (User Datagram Protocol (UDP), Internet Control Message Protocol (ICMP)) is fairly straightforward, as they can be passed to any member. Likewise, low-frequency stateful requests (DNS, DHCP, ARP) can be handled by a single fleet member, broadcasting information required to all fleet members. Therefore, the remainder of this section discusses TCP, which is the dominant workload of the network stack and exposes the most challenging problems.

Since TCP flows are stateful they must be handled specially, demonstrating how fleet members can coordinate to handle a given OS service. When an application wishes to listen on a port it sends a message to the coordinator which adds state information associated with that application and port. Once a connection has been established, the coordinator passes responsibility for this flow to a fleet member. The coordinator also sets up a forwarding

notification such that other packets destined for this flow already in the coordinator's queue get sent to the fleet member who is assigned this flow.

While this approach potentially re-orders packets, as the forwarded packets can be interleaved with new input packets, TCP properly handles any re-ordering. Once the fleet member has accepted the stream, it notifies the network interface to forward flows based on a hash of the (source IP, source port, destination IP, destination port). Once this flow forwarding information has been updated in the network interface server, packets of this type are delivered directly to the fleet member and then the application. Note that these mechanisms occur behind libfos and are abstracted from the application behind convenient interfaces.

Figure 8 repeats the previous experiment with the network stack and shows again that the fleet service model comes out ahead against Linux, even when accounting for OS cores.

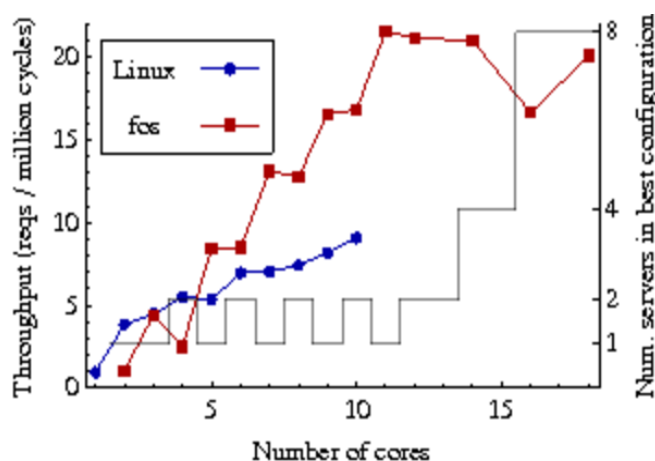


Figure 8: Comparison of Linux and fos network scaling

**4.7.4 File System and Block Device.** We have implemented a device driver server which interfaces with Xen's generalized block device and provides the low-level access to storage. We have also created a filesystem server which creates an ext2 filesystem on a block device by communicating with the block device server. File systems are not a primary research target for fos, so our filesystem is very much a placeholder. It is parallel and distributed for read-only workloads, but only supports a single server for write workloads.

**4.7.5 Process Management.** Using the generic dPool data structure that we developed, we have implemented a new parallel system service, the Process Management Service (PMS). The PMS is in charge of coordinating startup and shutdown of processes, including the tasks of allocating memory and computational resources, assigning process identification numbers (PIDs), loading program code and data, and de-allocating resources on shutdown. To accomplish these tasks, it primarily communicates with other system services including the name server, physical memory allocator, and filesystem. However, it manages PIDs internally and uses a dPool to store them. Previously dPool was only used by the physical page allocator; this quarter, a different programmer used it to get parallel version of the PMS working quickly and easily. This demonstrates that the dPool interface is both general and easy to use, as designed. We believe that it will also be useful in other system services and even user applications.

The process management server handles a variety of tasks including process creation, migration and termination in addition to maintaining statistics about process execution. Process migration is one technique we are currently developing to address fault-resilience and fault-tolerance in fos. This technique will allow us to move system servers and application processes between cores within the same machine and between machines within the cloud when a hardware fault is detected. This technique will, in turn, improve the overall robustness of the system.

**4.7.6 Cloud Management.** We have a cloud manager interface server that can communicate with cloud infrastructures like Eucalyptus and Amazon's EC2 to request additional VMs. This leverages the network stack to communicate with the outside cloud management infrastructure.

## **4.8 Distributed Data Structures**

**4.8.1 dPool.** We designed and implemented a scalable, distributed data structure called a dPool that is used to implement parallel system services. Parallel system services are provided by collections of cooperating server processes (referred to as fleets) whose members are distributed throughout a system. One of the major challenges of creating fos system service fleets is sharing state between the different fleet server processes. Because all of the processes in a fos fleet only communicate via message passing, the fleet programmer in order to effectively share state needs to partition the data and devise a manner to use messages to keep the state consistent across server processes. One way to address the challenge is to factor out the shared state into a distributed data structure which manages all of the communication to keep the state consistent. A distributed data structure created in such manner can then be used by different fleets in order to leverage the work of creating such a library.

The dPool data structure is designed to manage the data for a particular type of shared state: an unordered collection of elements. The shared state is encapsulated within the dPool data structure and the fos system programmer is simply presented with function calls to add and remove elements from the dPool. The programmer cannot request a particular element but just requests some element and the dPool can decide which element to return. The dPool data structure internally decides where to store the elements as well as sending and receiving messages to keep the shared collection of elements synchronized. This data structure is useful in situations where there is a single pool of essentially equivalent resources that need to be shared by multiple processes such as physical memory pages or process ID numbers. To facilitate its use by multiple different fos service fleets, a dPool provides a generic interface that can store any type of objects.

There are many options to consider when implementing a dPool data structure. The key issues to consider when implementing a dPool data structure are data placement and data rebalancing. The simplest choice is to store all the elements in a single location and send all requests for elements to this location. Obviously, this solution will not scale under heavy load. We have instead implemented distributed storage where the elements are partitioned and each piece is stored in a different fleet member. This scheme increases the available request rate and can be scaled by distributing the elements to more servers as demand increases. However, it introduces the problem of rebalancing when some servers may deplete their allocation of elements more quickly than others. Rebalancing can take place only when the supply of elements in a server is exhausted, but doing so introduces additional latency. fos makes use of spatial multiplexing of services to do re-balancing in the background. We have also implemented

four different schemes for rebalancing the elements among shards: bulk transfer, background pull, background push, and background push with estimation.

The normal Distributed Storage implementation requests an element from other shards only when it completely runs out of its own elements. The Bulk Transfer implementation behaves similarly but prefetches a block of elements when it needs to make a request. The Background Pull implementation contains a second thread in each server that occasionally wakes up and checks to see if the supply of elements in that shard is starting to run low. If the number of elements is below some threshold, it pulls blocks of elements from other shards in the background. The Background Push implementation also uses a background thread but takes the opposite approach and pushes elements to other shards when it notices that the local shard has a surplus of elements. However, this version pushes indiscriminately to other shards whether they need extra elements or not. The Background Push with Estimation implementation improves on this by maintaining an estimate of the number of elements in the other shards which is lazily updated to reduce communication traffic. Using this estimate, a particular server will only push elements to other shards that it estimates to have fewer elements than it does. This greatly reduces the possibility of two servers repeatedly pushing the same elements back and forth to each other.

**4.8.2 Key-Value Store.** The name service uses a fully distributed key-value (KV) store. It is completely replicated for read performance and fault tolerance. Consensus on updates to the key-value store is reached by a simple two-phase commit protocol.

This data structure has been implemented generically and can be used in other services as well. In the current implementation, however, most services choose to use the name service to store shared state rather than incorporate the KV store internally. That is, the name service works to redirect requests to the server that owns the object. This minimizes the need for replication of larger OS objects and improves performance.

We have begun initial exploratory work into other KV stores that will provide better write latency, limit replication to fewer nodes, and improve scalability. There are two main paths for this work: a cache-coherent, strongly-consistent variation, or an eventually consistent variation. Each are viable options under consideration, with the main trade-off coming in the semantics presented the programmer and additional performance cost for strong consistency.

## **4.9 Tilera Port**

We have also begun work on a port of fos to run on Tilera hardware. This primarily involves porting machine-specific code within the microkernel. Most services other than the process management service, which manages hardware-specific structures like page tables, are machine independent, including the name service and messaging system. Once the microkernel work is complete, this should quickly translate to a complete port of the full system. However, this work will not be completed within the timeframe of the award.

## **5.0 CONCLUSIONS**

Current operating system designs will not be able to cope with the future of multicore systems. The differences between managing a couple and several thousand cores in these systems is so drastic that the entire design of operating systems must be rethought.

fos is designed deal with this problem by factoring OS services into userspace processes which run on separate cores from applications. Doing so has several scalability and performance advantages: primarily increased parallelism and decreased cache pollution between applications and system services. Other advantages include fault tolerance, self-awareness, and elastically scaling services in response to changing demand.

A working prototype of fos has been implemented as a paravirtualized OS under the Xen hypervisor. The prototype features a messaging layer on which several system services have been implemented. Implemented services include: naming, page allocation, file system, network stack, and process management. fos has also been extended to support dynamically adding new machines to a running fos image. This allows the capacity of the system to be increased at runtime.

## 6.0 REFERENCES

1. Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," University of California, Berkeley, Technical Report No. UCB/EECS-2006-183, December 18, 2006.
2. Borkar, S., "Thousand Core Chips - A Technology Perspective," *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE* , pp.746-749, June 2007
3. Borkar, S., "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE* , vol.25, no.6, pp. 10-16, Nov-Dec 2005
4. Mukherjee, S.S.; Emer, J.; Reinhardt, S.K., "The soft error problem: an architectural perspective," *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on* , pp. 243-247, Feb 2005
5. J. Kneschke. Lighttpd. <http://www.lighttpd.net/>
6. B. Fitzpatrick. Memcached. <http://memcached.org/>
7. SQLite, <http://www.sqlite.org/>
8. SPLASH. <http://www.capsl.udel.edu/splash/>
9. C. Bienia, S. Kumar, J.P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In PACT, 2008.
10. B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175 – 198, May 1991
11. A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhanian. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, 2009.

## **LIST OF SYMBOLS, ABBREVIATIONS AND ACRONYMS**

2D: two dimensional  
3D: three dimensional  
ACM: Association for Computing Machinery  
API: application programming interface  
ARP: Address Resolution Protocol  
C: A programming language.  
CMOS: complementary metal-oxide-semiconductor  
DNS: Domain Name System  
DHCP: Dynamic Host Configuration Protocol  
EC2: Elastic Compute Cloud  
ICMP: Internet Control Message Protocol  
IEEE: Institute of Electrical and Electronics Engineers  
IP: Internet Protocol  
ISA: instruction set architecture  
KV: key-value  
lwIP: A light-weight open-source IP stack implementation.  
OS: operating system  
PID: process identification number  
PMA: physical memory allocation  
PMS: process management service  
POSIX: Portable Operating System Interface  
RPC: remote procedure call  
SMP: symmetric multi-processor  
TCP: Transmission Control Protocol  
UDP: User Datagram Protocol  
URI: uniform resource identifier  
URPC: user-level remote procedure call  
VM: virtual machine